Systems programming

12 – Shared data

MEEC LEEC MEAer LEAer MEIC-A
João Nuno Silva

TÉCNICO LISBOA

Bibliography

- Sun Micorsystsmnes, Multithreaded Programming Guide, chapter 4
-
- Michael KerrisK, The Linux Programming Interface, Chapter 30

Shared data in threads

- Shared variables
  - Global variable in C
  - Memory region allocated
- Shared data should be avoided
- But sometimes is required
  - Two thread access (modify) same variable

Data access

- Access to data in memory is not atomic
  - It is necessary to load, process and store
  - Even without caches
  - Even on a single computer
- The simple C operation (i++) is complex

General	MIPS	x86	ARM

Load Reg ← Mem
Increment register
Store Mem ← reg
	lw $t0, label
	addiu $t0, $t0, 1
	sw $t0, label
		Mov ax, label
		Inc ax
		Mov, label, ax
			LDR R2, [label]
			ADDS R2, R2, 1
			STR R2, [label]

Instruction ordering

- Single-threaded application
  - Order of instructions is the order of C Code
- Multi-threaded application
  - Order of instructions is one of the possible combinations

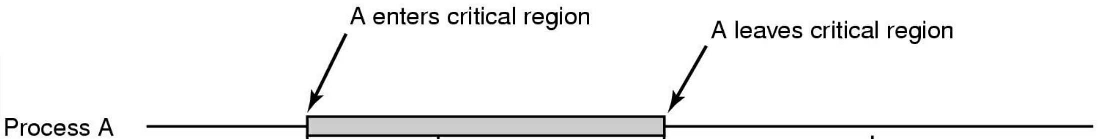
Main	Thread 1	Thread 2
N = 0 pthread_create		
	Load R1 ← N R1 ← R1 +1 Store N ← R1	Load R1 ← N R1 ← R1 +1 Store N ← R1

Thread synchronization

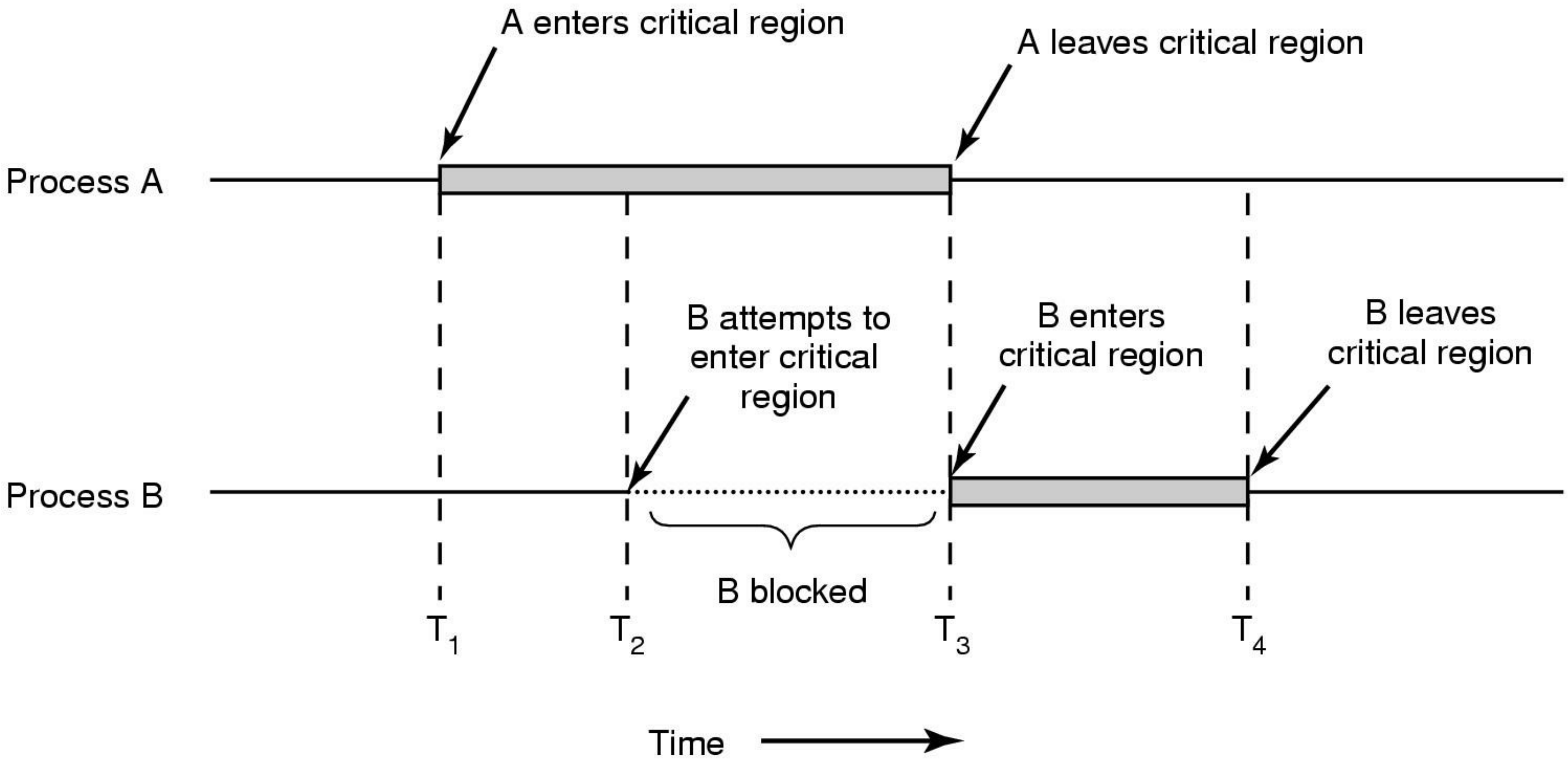
- Threads execution order
  - defined by the programmer not efficient
  - Multiple combinations
- Protect accesses to shared variables
  - Define critical region
  - Only one such critical region is executed at a time
    - Only one thread is inside such section
- Implement mutual exclusion

Critical region

- Piece of code where resources are shared
  - That can should be executed by one tasks at a time
  - Is delimited by read/write instructions to the shared resource
- If a task is inside a critical region
  - Other task trying to enter should be blocked



Critical region



Critical region

- Requirements to be satisfied
  - Mutual exclusion
    - Only one task can be inside the Critical region
  - Progress
    - A task inside the Critical region can not block other task from entering
  - Limited wait
    - A task should wait a limited amount of time before entering
- A task should remain inside the Critical Region for a limited time
- The speed and number of processor is undefined
  - Should work for any combination of CPU/tasks

Mutual exclusion

- Mechanism that assures that at most one tasks is inside a critical region
  - No more that one task is executing the critical region code
- All other tasks are
  - Running non critical code
  - Blocked
- Consequences of mutual exclusion
  - starvation (live-lock)
    - A task is able to be executed, but is never scheduled
  - deadlock
    - Due to coding problem, several tasks are waiting for being unblocked by other tasks that are in the same state

Critical region / Mutual exclusion

```
do {
   non_critical_code()
   EnterRegion()
CriticalRegion(shared_data)
   LeaveRegion()
   non_critical_code()
} while(X);
```

zero or more tasks

Zero or one tasks

All other tasks

Locks

- The simplest mechanism to ensure
  - mutual exclusion of critical sections

- Spin locks
- Busy waiting is inefficient
  - NOPs use energy
- Requires special instructions
  - While is a critical region
  - Test-and-set
  -

```
while (lock == 1) {}
lock = 1;
// enter crit reg
criticalRegion()
lock = 0;
// leave crit reg
```

Mutual exclusive locks - mutex

- OS object that
  - have 2 values (0 or 1)
  - Have a tasks waiting list
  - Is managed by the operation system
    - To suspend and run tasks
- A Mutex can be in the following states
  - Unlocked – no tasks inside the critical region
  - Locked by a single task – the one in the critical region

Mutex usage

- After defining a critical region
  - Assign it a mutex
  - Initialize a mutex
- Each thread
  - Call mutex_lock when entering the critical region
  - Call mutex_unlock when leaving the critical region
- Mutex_lock
  - Tasks can be placed in a list, waiting for a mutex_unlock
- Mutex_unlock
  - Can only be called by the thread inside the critical region

Mutex usage

- Good programming

```
mutex_lock(m1)
CriticalRegion()
mutex_unlock(m1)
```

  - Small critical region
  - Same code on all tasks

- Bad programming
  - Long critical regions
  - mutex_lock(m1)
    - Inside critical region
    - Without unlock
  - mutex_unlock()
    - Not owner
    - Outside Critical Region

POSIX mutexes

- POSIX mutexes are associated to Pthreads:
  - include file **`#include <pthread.h>`**
  - Compile with **`-lpthtread`**
- The corresponding data type is
  - **`pthread_mutex_t mux;`**
- A mutex should be initialized before used
  - **`mux=PTHREAD_MUTEX_INITIALIZER;`**
- A mutex is destroyed by calling:
  - **`int pthread_mutex_destroy(pthread_mutex_t *);`**

POSIX mutexes

- mutex locking
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - Blocks thread until resource is available/can enter critical region
  - Returns when task enters critical region
  - Returns 0 in case of success
- Mutexes should be locked for the minimum amount of time
- Mutex unlock
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - Returns 0 in case of success
  - Allow other thread to enter region
  - Unblock a thread from pthread_mutex_lock

POSIX Spin Locks

- Spin locks are a low-level synchronization mechanism
  - suitable for tiny critical regions
- When the calling thread requests a spin lock that is already held by another thread
  - the second thread spins in a loop to test if the lock has become available.
- When the lock is obtained it should be held only for a short time,
  - as the spinning wastes processor cycles.

POSIX Spin Locks

- POSIX spin locks are associated to Pthreads:
  - include file #include <pthread.h>
  - Compile with -lpthtread
- The corresponding data type is
  - pthread_spinlock_t *lock;
- initialization
  - int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
- Destruction
  - int pthread_spin_destroy(pthread_spinlock_t *lock);

POSIX Spin Locks

- locking
  - `int pthread_spin_lock(pthread_spinlock_t *lock);`
- unlocking
  - `int pthread_spin_unlock(pthread_spinlock_t *lock);`

POSIX Spin Locks

- A programmer must be exceptionally careful with
  - the code, system configuration, thread placement, and priority
- If a thread creates deadlock while holding a spin lock
  - It will spin forever consuming CPU time.
- If a thread is scheduled off the CPU while it holds a spin lock
  - other threads will waste time spinning on the lock until
    - the lock holder is once more rescheduled
    - and releases the lock